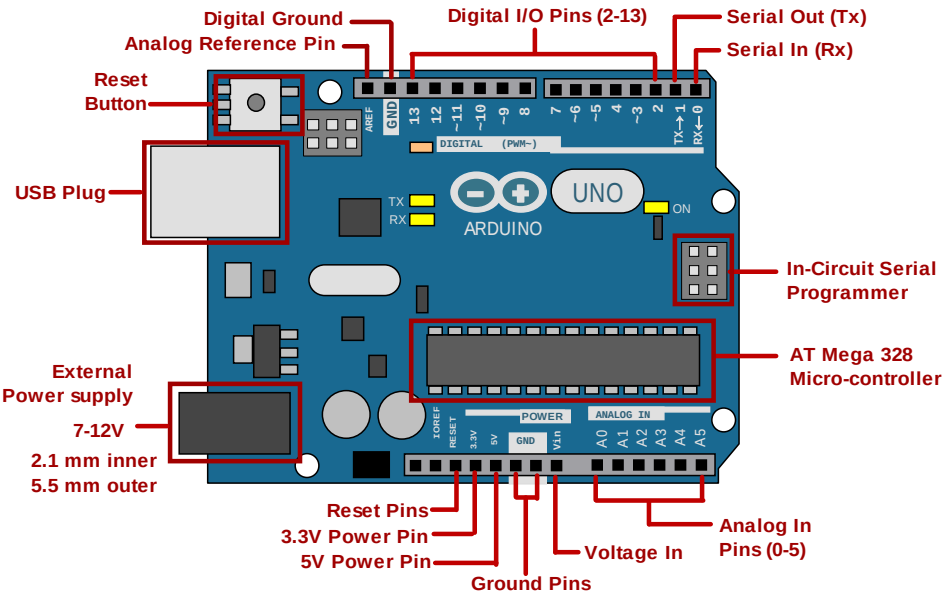
ARDUINO GUIDE

CONTENTS

CHAP 1 - INTRO



Micro-controllers (or micro-controller units, MCU's) are computers that are designed to interact with the physical world (e.g., sense light, sense sound, turn LED's on/off, control motors). There are many brands but we will work with ARDUINO.

Arduino has become vey popular in the hobbyist and academic communities. YouTube shows a great number of cool projects that people have done. There are many websites with sample code and projects to get you going. Arduino has many different models, but we'll work with UNO R3 or MEGA 2560 R3.

Arduino is an open-source hardware and software platform. Thus, other manufacturer produce "Arduino compatible" boards (e.g., Elegoo, Inland, Aesop, etc.). These boards work fine and they cost less. Arduino software (called Arduino IDE, or Integrated Development Environment) is free to download and use. Arduino programs are called SKETCHES. You can also expand Arduino's capability by purchasing add-on circuit boards called SHIELDS. For instance, there are shields for adding Bluetooth or radio-control communications, motor control, and much more...

1.1 ARDUINO HARDWARE & SPECS

UNO

MEGA

(specs need not be memorized)

1.1.1 UNO

1. Processor – ATmega328P, Clock speed: 16 MHz
2. Storage: flash (program storage): 32 kB; SRAM (store/manipulate variables while program runs): 2 kB;
3. EEPROM (long-term info): 1 kB
4. Input voltage (7-12V), I/O pins (14), current/pin (20 mA), Outputs: 3.3V (50 mA max?), 5V (0.8A max?)
5. PWM (8-bit, or 0 - 255, 6 of 14 pins, at 490 Hz, except 5 & 6, at 980 Hz),
6. ADC (8 channels, 10-bit (0 - 1023) over 0 - 5 V range)

1.1.2 MEGA

1. Processor – ATmega2560, Clock speed: 16 MHz
2. Storage: flash (program storage): 256 kB; SRAM (store/manipulate variables while program runs): 8 kB; EEPROM (long-term info): 4 kB
3. Input voltage (7-12V), I/O pins (54), current/pin (20 mA); Outputs: 3.3V (50 mA max?), 5V (0.8A max?)
4. PWM (8-bit, or 0 - 255, 15 of 54, at 490 Hz),
5. ADC (16 channels, 10-bit (0 - 1023) over 0 - 5 V range)

1.1.3 BOTH

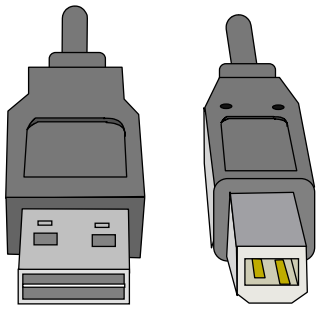
Power plug – barrel jack, 2.1 mm inner, 5.5 mm outer

CURRENT capability. 40 mA/pin (really closer to 20-25 mA), 200 mA total all pins.

PROCESSOR - Arduino uses the 16-MHz ATmega processor - not strong enough to do desktop computer work or graphics. But it IS strong enough to do things like light up lights, collect sensor data, and control motors. (note - if you need it, there's Arduino Due at 84 MHz for ~$48).

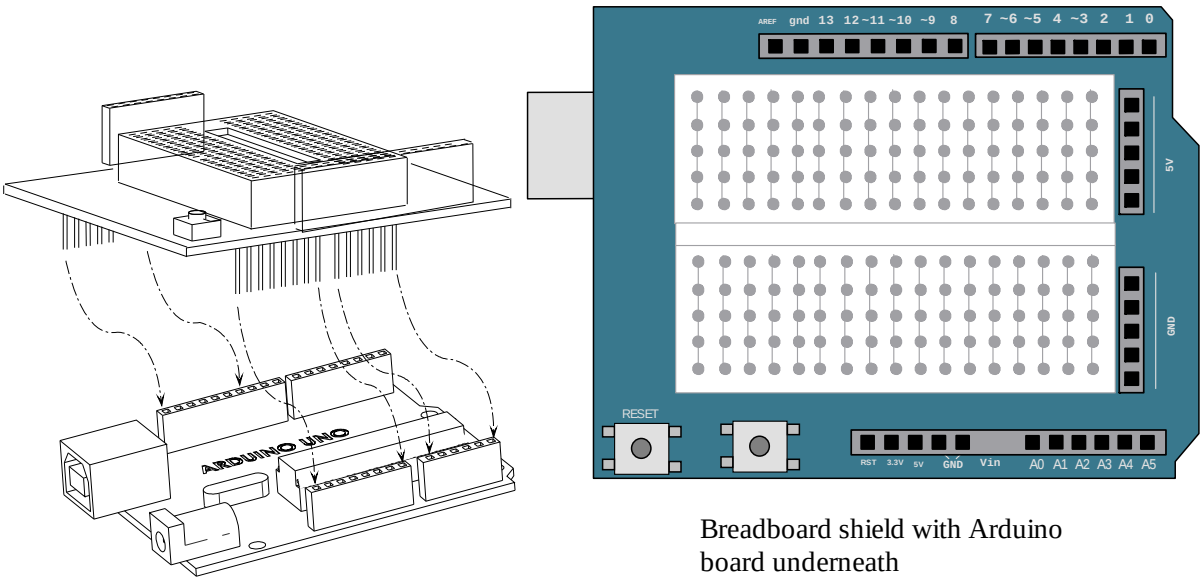
USB CABLE - Arduino uses a USB standard A-B cable. Arduino-compatible brands may use a different type of USB cable (eg, some use micro-USB).

SHIELDS - **SHIELDS** are add-on circuit boards that expand Arduino's capability. Their "form factor" (ie - they are sized) that allows them to stack onto the Arduino board. For instance, there are shields for adding Bluetooth, Ethernet, motor control, and much more.



USB STANDARD A-B CABLE

BREADBOARD SHIELDS - many Arduino kits will come with a breadboard shield. The shield is stacked on top of the Arduino. The shield's only purpose is to provide a small breadboard onto which you can build a simple circuit, but it also pulls the I/O pins up to the shield to make connections easier



Breadboard shield with Arduino board underneath

1.2 WHERE TO BUY

Arduino's are available in electronics stores (Microcenter in Tustin, Frys in Fountain Valley, Best Buy) or online (amazon, robotshop, adafruit, sparkfun).

1.3 ARDUINO SOFTWARE

The Arduino software is called the **Arduino IDE** (Integrated Development Environment). The software is free to download and use. Arduino programs are called sketches.

Get Arduino software (IDE) at arduino.cc (or do a search on "Arduino IDE").

TO KNOW FOR EXAM:
What are MCU's? What brand and models will we work with? What is the Arduino software called? What are Arduino programs called? What are shields?

CHAP 2 - ARDUINO IDE

The software for running Arduino is the Arduino IDE (Integrated Development Environment). It is free to download and use (just search: Arduino IDE). The software is available for PC, MAC, and Linux systems.

The Arduino IDE includes

1. Hardware drivers – allows your computer to recognize the Arduino hardware
2. Programming editor – to type your programs
3. Compiler – to translate your programs into code that the Arduino hardware can execute
4. Arduino functions – there are MANY pre-written functions to use and access the board's capabilities.
4. Example code – there are MANY pre-written programs to do common tasks (light LED, drive servo, etc.)

Arduino is programmed in the C++ programming language.

2.1 Download

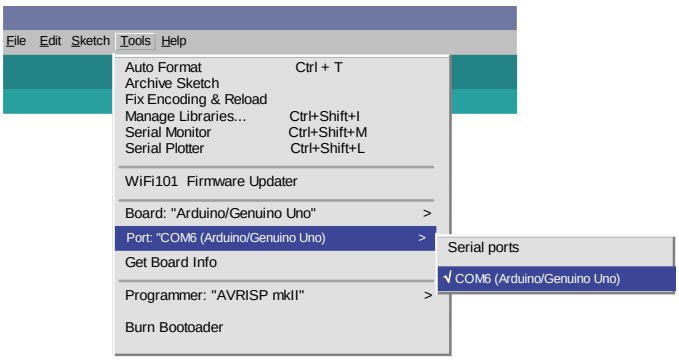
Download and install the software. Open the software (in Windows, "Arduino" in the search). Select it. Plug the Arduino board into a USB port using USB cable. Arduino will automatically draw its power from the USB connection. There is an option for drawing power from an external power supply.
Wait for the OS to recognize the connected Arduino.

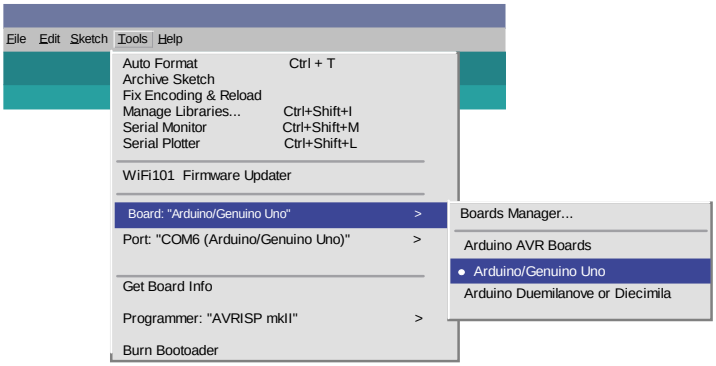
2.2 Board & Port (IMPORTANT!!!)

Go to File > New. A new window opens with the setup and loop functions already there.

Before doing anything, ensure the proper PORT and BOARD are selected! This may need to be done every time the board is unplugged and re-plugged in, or if you open a new program file. These steps are definitely needed if you are using another board (even if it is the same model as the first board).

(Tools > Ports > Com XX)
(Tools > Board > Arduino Uno (or Mega))



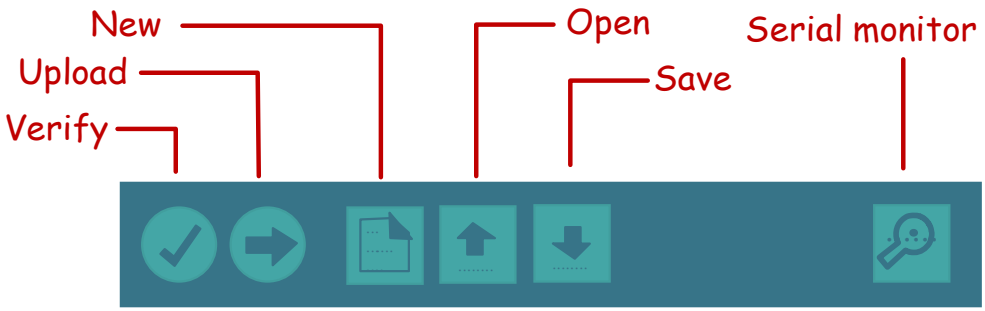


You may need to do these checks each time you re-plug in a board, plug in a new board, or start up a new program. If you receive an error indicating your program could not be uploaded to the Arduino, this is probably the solution!

2.3 Writing Arduino Programs

Arduino programs must include the **`setup()`** and **`loop()`** functions. The setup( ) function is executed first and it is done one time. The loop( ) function is done after setup( ) and it is executed repeatedly until Arduino is unplugged. Arduino files are *.ino. They must reside in a folder of the same name (the software usually does this automatically). Arduino file names should never have a SPACE.

You can start a new program (File > New). A new window opens with setup() and loop() already there.



Writing Arduino code amounts to defining what the setup() and loop() functions do.

Once written, you can Save, Verify, or Upload your code.

Verify means to compile your code.

"Upload" will upload your code to the board, but the software will first automatically compile your code. Any compile-time errors must be fixed or the software will not upload the code to the board.

Once uploaded, the prior program loaded in Arduino is REPLACED by the new code.
Arduino will only store one program at a time.

2.4 Arduino Program Menu

- Verify – compile your program
- Upload – compile and send the program to the Arduino board.
  - The RX & TX lights should flash, indicating communication between the computer and Arduino.
  - Upon successful upload "Done loading" appears in the status bar.
- New – create new program
- Open – open existing program
- Save – save current program
- Serial monitor – open the serial monitor

2.5 Keyboard Shortcuts

Keyboard shortcuts are handy and faster.

ctl-s	Save
ctl-r	Verify/compile
ctl-u	Upload your program to the board
ctl-shift-m	Show Serial monitor
ctl-shift-l	Show Serial plotter
ctl-t	Auto-format

Preferences file – allows you to customize the Arduino software

2.6 Compiling & Errors

Press VERIFY to compile your completed code. The compiler will check for errors first. If it finds errors, it will display a warning at the bottom of the window (in orange/red). These errors must be fixed. The compiler warning may seem cryptic and not helpful. Unfortunately, compilers are not smart enough to be specific about your error (it won't say "you forgot a semi-colon" on this statement).

If there are no errors, the compiler will translate your code into machine language that the CPU can execute.

Then you must UPLOAD your code (i.e., send it to the board). Pressing UPLOAD will compile the code and upload (so you can skip VERIFY if you want). Arduino can only hold ONE program at a time. Once uploaded, the Arduino will start functioning right away.

TO KNOW FOR EXAM:
What is included in the Arduino software? Board and port checks (why do it?)
What are: verify and upload (identify them on the software toolbar)?

CHAP 3 - PROGRAMMING

Arduino is programmed in the C++ programming language - possibly the most important programming language for engineers. C is a sequential (commands are executed in order), block-structured (more on this later), procedural (it uses functions), free-form (ignores spaces and hard returns) language. C++ adds the ability to create so-called CLASSES (i.e., object-oriented programming). C and C++ are COMPLIED languages.

Best practices for programming include:
1. Design programs to be logical and step-by-step.
2. Break down code into small modular elements.
3. Code should be broken down and organized into simpler functions (small blocks of code that do one thing).

You don't have to write everything from scratch. The Arduino software includes many libraries that have many pre-written functions. There are many more available online.

Refer to the PROGRAMMING GUIDE for more detailed information on programming the Arduino in C/C++. You can also buy a book on programming in C/C++ or Arduino programming, or you can take a class (E.g., CMPR 120 and 121 teaches C++).

3.1 First Program (Blink)

A new Arduino program will have empty setup ( ) and loop ( ) functions. Programming Arduino amounts to filling in what these functions do. When doing this, we will often CALL pre-existing Arduino functions. We can also define OTHER our own custom functions.

Often, the first Arduino program we write will be to blink the surface-mounted LED on the Arduino board. This LED is connected to pin 13 of the board. You can write this yourself or you can use example code included with the Arduino software (File > Examples > 01.Basics > Blink). There is a bunch of pre-written code included with the Arduino software.

The code below "calls" several functions (pinMode(), digitalWrite(), delay()). Those functions are defined somewhere else. The "argument" for the delay() function is the amount of time (in milli-seconds) that the processor pauses. We use it to control duration. Try playing with these numbers and see how it affects the blinking of the LED.

CODE

```
// blink.ino ----------------
//LED 13 on Arduino blinks repeatedly (no inputs) using delay() fcn


int ledPin = 13;          // declare global variable, pick pin 13


void setup()
{
   pinMode(ledPin, OUTPUT);      //set ledpin to be an output
}

void loop()
{
  digitalWrite(ledPin, HIGH);         // LED on
  delay (100);                        // hold it for this many ms
  digitalWrite(ledPin, LOW);          // LED off
  delay(100);                         // hold it
}

// end ----------------------------------------------
```

3.2 Neat, Commented Code

It is fastest to write you code with best practices. This includes making your code readable using comments, white spacing, and proper indentation. You can auto-indent/format Arduino code by pressing "ctl-t" (or Tools > Auto-Format).

Comments - using lots of brief comments, especially single-line comments that explain what your code does
White space - code that has lots of "white space" is easier to read. For instance always put a blank line before the start of a function definition. Nearly always insert a hard return after an executable statement (with a ";"). Blocks (the stuff in between braces " { } " should be indented).

3.3 Debugging

The compiler may find errors in your code. If that happens you must correct the error in a process called DEBUGGING. Even if your code compiles that does not mean that it will work as intended.

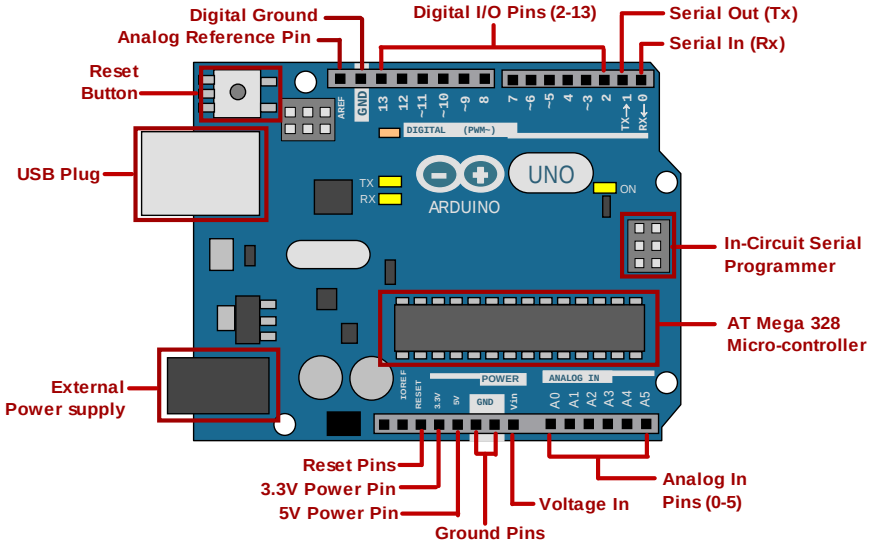
It is important to keep your program neat, organized, and commented. Otherwise debugging will take much longer! Shortcuts never pay! Disorganized, poorly-space, poorly-commented code takes MUCH longer to debug. It is FASTER to be neat, organized, and commented.

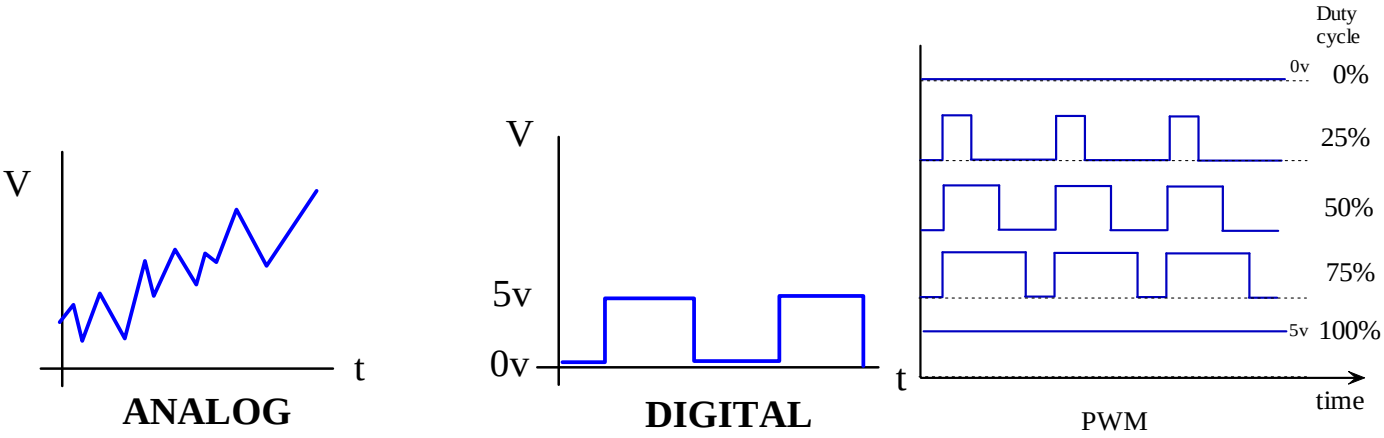
Write variables to the Serial Monitor to display their values. This is a good way to debug your code.

The most common programming errors are errors in syntax (form).
Common programming errors include:
1. Unmatched parentheses, brackets, or braces
2. Variables meant to be the same but are not, for instance due to capitalization (myVar3 vs. myvar3)
3. Undeclared variables
4. Missing semi-colons

TO KNOW FOR EXAM:
Arduino is programmed in what language?
What is debugging?
Code should have what characteristics? (neat, commented)
Common data types used?

CHAP 4 - I/O PINS & SIGNALS





Computers (including microcontrollers) communicate using electrical signals. Arduino uses I/O (input-output) pins. The pins can be configured as either INPUT (reading a signal) or OUTPUT (sending a signal out to control something). There are digital signals and analog signals (see below). Arduino can do digital input or output. Arduino can also do analog input. Arduino does not have true analog output. Instead it uses pulse-width modulation (PWM). See below.

4.1 Electrical Signals

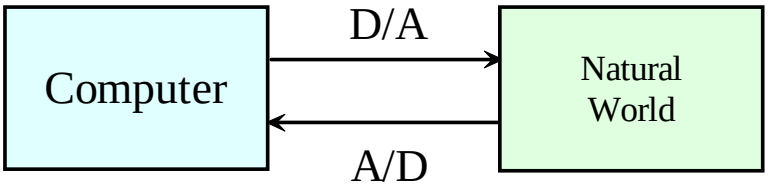
Information is transmitted using electrical signals. There 2 basic types of electrical signals:.

1. Analog – can vary continuously (natural world is analog) like a ramp.
2. Digital – sequence of discrete values (computer world is digital) like steps.

Moreover signals are either input or output.

Digital signals can be configured to take on any number of steps, but modern computers use BINARY digital signals where there is only 2 values: 0 (LOW or 0 VDC) or 1 (HIGH or 5 VDC). The natural world tends to use analog signals. Thus, any I/O communication between the computer and the natural world requires signal conversion.

Digital-to-analog conversion (D/A) - allows the computer to send signals out to the natural world.
Analog-to-digital conversion (A/D) - allows the computer to receive signals from the natural world.



D/A and A/D conversion can occur with different RESOLUTIONS. This is determined by the number of bits used in the conversion (e.g., 8-bt conversion). Take the example below with 2-bit conversion. With binary digital signals you can either have 0 or 1. Think of that as the number of letters in your alphabet. With TWO-bit conversion, think of the number of bits as the number of letters in a word. So how many possible words are possible with 2 letter types, and only words having 2 letters each?

2's	1's	Total
0	0	0
0	1	1
1	0	2
1	1	3

In the example above, there are 4 possible words valued 0 to 3. This means there are 4 possible signal levels. This is pretty bad resolution. We can add resolution by adding more bits. Let' try 3-bit resolution. See the table below. You can see there are 8 levels now.

4's	2's	1's	Total
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

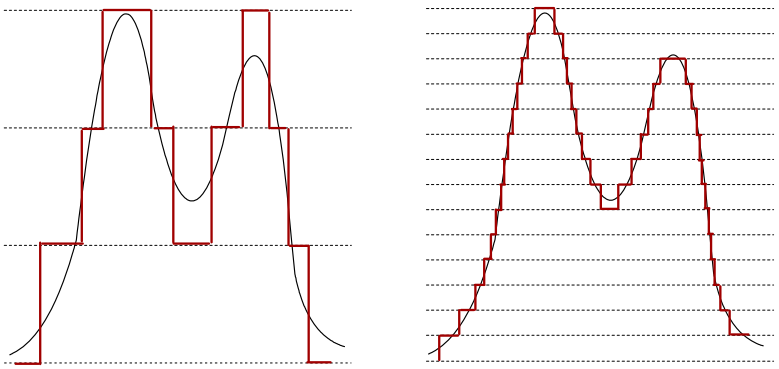
The formula for getting the levels for any number of bits is:

$$2^{n}$$

n ← # bits

2 ← binary system (2 values)

See the figure below comparing 2-bit A/D conversion vs. 4-bit resolution. The stepped trace is the psuedo-analog signal. See how the 4-bit resolution trace looks much closer the analog signal.



2-bit resolution 4-bit resolution

Arduino's analog input uses 10 bits, so its resolution is 1024 levels (or 2^{10}).

DAQ (data acquisition) hardware used in industrial robotics will sometimes use 24-bit conversion. This produces 16,777,216 levels (or 2^{24}). When you have more levels, you have better resolution and the "psuedo" analog signals appears closer to the real analog signal which is usually desired.

4.2 Digital I/O

Most pins on Arduino are digital I/O (input or output) pins. These "pins" are configured as input or output using the functions below.

```
pinMode(pin, val);   // read a digital pin, val = INPUT or OUTPUT
```

These are commanded using the functions below.

```
digitalRead(pin);          // read a digital pin
digitalWrite(pin, val);    // write a value to a pin, val = HIGH or LOW
```

4.3 Analog Input

Arduino has a few ANALOG INPUT pins which receive analog voltages in the 0 to 5 V range. Internally, Arduino converts these to digital signals using 10-bit ADC (analog-to-digital conversion) which means the 0 to 5 V range is chopped up into 1024 levels. The function below returns an integer between 0 and 1023.

```
analogRead(pin);           // read an analog pin, returns a value 0 to 1023
```

4.4 PWM Output

Arduino does NOT have any true analog OUTPUT pins. Instead, Arduino uses pulse-width modulation (PWM) to emulate analog output. On Uno, these pins are denoted with a "~" or "PWM". PWM signals are still digital (only 0 and 5 V is possible), however they remain high for only a certain amount of time and is otherwise low (the percentage of time high is called DUTY CYCLE). Arduino PWM has 8-bit resolution, so range of PWM values is 0 (LOW all the time) to 255 (HIGH all the time). This is 256 values. The frequency of PWM is 490 Hz, though 2 pins (pins 5 & 6 on Uno) run at 980 Hz.

```
analogWrite(pin, val);     // pin = PWM pin, val = 0 to 255
```

Modern dimmer switches also use PWM. If you dim your lights to, say, half brightness, what is really happening is that the dimmer switch is switching the light on and off very quickly (half the time on, half off). The switching rate is far too high for our eyes and brain to see (our limit is somewhere around 50 or 60 Hz). So instead, our brain interprets what it is seeing as a dimmed light!

4.5 UNO Pin Specs

Digital I/O pins (14 pins, 20 mA max) either input or output, transmit HIGH (5V) or LOW (0V) values.

Analog input - 8 pins, 0 - 5 V range, 10-bit resolution (0-1023)

PWM pins - 6 pins, 8-bit resolution (256 width values), ~ 490 Hz frequency (pins 5, 6 = 980 Hz)

4.6 Arduino I/O Pin Functions

`pinMode (pin, mode);`	configure I/O pin as either INPUT or OUTPUT
`digitalWrite (pin, mode);`	output a HIGH (5V) or LOW (0V) to a pin (mode can be 0 & 1)
`digitalRead (pin);`	read I/O pin and returns HIGH or LOW
`analogRead (pin);`	read analog input pin (returns 0 to 1023, mapped from 0 to 5 V)
`analogWrite (pin, value);`	write pulsewidth to PWM pin (value = 0 to 255)

TO KNOW FOR EXAM:
Analog vs. digital signals (properties)
D/A and A/D conversion (resolution, resolution equation)
Resolution - how compute # level (2 to the "n" equation).
Arduino functions for digital in/out, analog in, pwm out
AnalogRead returns (values from 0 to 1023, corresponding to 0 to 5V)
AnalogWrite (pin, value) (value range: 0 to 255)

CHAP 5 - SERIAL MONITOR

The SERIAL MONITOR for Arduino IDE is a separate pop-up window that acts as a terminal that communicates by receiving and sending serial data. To open the serial monitor window: Tools > Serial Monitor (or ctl-shift-m). The serial monitor is somewhat analogous to using the console output for regular C (cout command). The Arduino must be plugged into the computer when using the Serial Monitor. Users can use the serial monitor to display the values of variables in your program. This can be a valuable tool in debugging your program. Students who are experienced with C programming can think of the Serial Monitor as a type of console output (recall cout, cin, etc.)

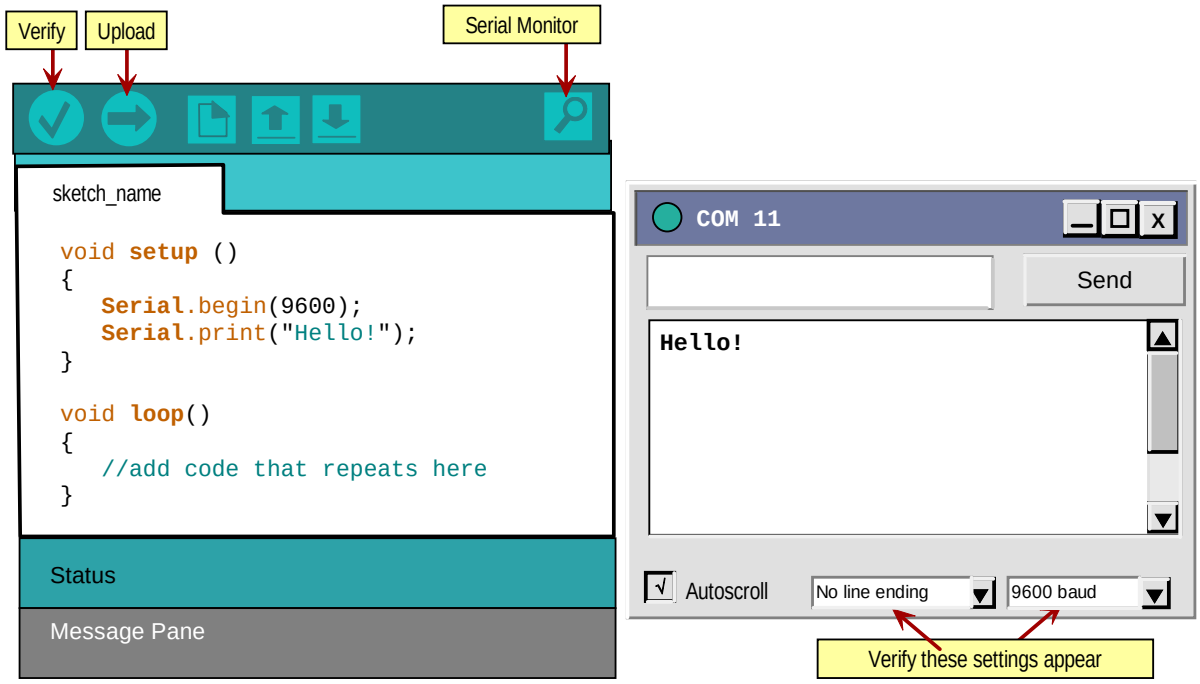
Here are some functions related to the serial monitor.

Serial.begin causes Arduino to set aside some memory for characters coming in from the Serial Monitor. This memory is typically called a SERIAL BUFFER and it stores the ASCII values from the Serial Monitor. It is common to set the baud rate to 9600 (so, **Serial.begin(9600);** is placed in setup function)

Serial.read()	gets a character from the buffer
Serial.available()	will tell you how many characters are in the buffer.
Serial.print()	writes to the serial monitor
Serial.println()	writes to the serial monitor on a new line

If displaying a decimal value you can control the number of places displayed using:

Serial.print(1.123, 2); displays "1.12"



SERIAL MONITOR EXAMPLE

Below is a simple program that adds 2 numbers. We declare and assign values to **a** and **b**. The program then adds **a** and **b** and assigns the result to c. The result is sent to the serial monitor.

All of the statements we write are in the setup function (this defines what setup does). Nothing is written in the loop function (nothing is repeated here). However, we must write the blank loop function definition or the program will not compile.

```
//FirstProgram.ino ------------------------

void setup ()
{
  Serial.begin(9600);       //set baud rate to 9600 to use serial monitor

  int a = 3;        //initialize variable "a" & place value of 3 in it
```

```
  int b = 4;        //initialize variable "b" & place value of 4 in it
  int c = a + b;          //add values in a & b, place result in c

  Serial.print(c);  // print value in c in serial monitor
}

void loop ()  // must still type this even though its empty
{
}

//end ------------------------------------
```

The statement **`Serial.print(c)`**; will display the value held in variable c. The Serial Monitor will be a valuable tool in helping DEBUG or FIX programs that have errors.

The Serial.print function will print on the same line. If you want the next line to appear on a new line you should instead use Serial.println.

SERIAL PLOTTER

Associated with the Serial Monitor is the **Serial Plotter** (Tools > Serial Plotter). The serial plotter is able to plot variables in real time as time goes on. The Arduino IDE plots variables that have been written to the Serial Monitor.

The plotter can also plot multiple lines. In code, you must write spaces between the 2 variables to get the plotter to plot separate lines (**`Serial.print("   ");`**).

```
Serial.print(data1);
Serial.print("   ");      //separator needed to plot both data1 & data2
Serial.println(data2);
```

NUMBER OF PLACES

You can control the number of places after the decimal using the statement below

```
Serial.print(x, numberPlaces);
```

TO KNOW FOR EXAM:
What is the serial monitor? How is it used? How do you get it going?
How do you displaying a variable in it?
How do you display the serial monitor?
What is the serial plotter?

CHAP 6 - TIMER

Arduino UNO has a couple of timers built onto the board. Sometimes it is helpful to access these timers in our program.

`delay (ms);`	pauses the processor for so many milliseconds
`delayMicroseconds();`	pause processor for so many micro-seconds
`millis();`	returns the number of ms that have elapsed since Arduino started (e.g., returns 1000 if executed 1 second after Arduino started)
`micros();`	returns the number of micro-seconds that have elapsed since Arduino started (e.g., returns 1,000,000 if executed 1 second after Arduino started)

You should assign millis() or micros() to unsigned long.

Millis will overflow a "long" in 24 days, an unsigned long in about 48 days.
Micros will overflow a "long" in 35.7 minutes, an unsigned long in ~ 61 minutes.

Some articles indicate that you do not need to worry about overflowing Arduino. It will not crash the micro-controller. They indicate that you get "roll over", where the value of the variable cycles back to 0. Usually the user calls millis or micros in order to check time elapsed versus some reference time. As long as you subtract the difference, you likely will not approach the data type limit.

```
//example
int telapsed
telapsed = millis() - tref;
```

The delay() function essentially PAUSES the processor for the specified period. During that period, the state of the I/O pins is retained.

TIMER CODE

```
//timer_code.ino ----------------
//simple eg of using arduino timer fcns

unsigned long t1, t2;

void setup ( )
{
}

void loop ( )
{
  t1 = millis();          //returns int = # ms since arduino started
```

```
  t2 = micros();          //returns int = # us since arduino started
}
```

TIMER FUNCTIONS

```
millis();
micros();
delay();
delayMicroseconds();
```

TO KNOW FOR EXAM:
Know the 4 timer functions (millis, micros, delay, delayMicroseconds)
How are they used? What do they return?

CHAP 7 - BLINK NO DELAY

You have seen the delay() function used in the blink exercise to control durations. This function causes the CPU to stop doing anything for a period of time. The present state of the pins remains unchanged during the delay period. This was very helpful and easy to use in the simple blink exercise. However **`delay()`** becomes a problem when programming Arduino to run MULTIPLE PROCESSES the same time. You may want the delay in one process, but not another process. It won't matter. Delay will stop almost all of them (some functions like the ones for controlling servos are designed to run in the background and are unaffected by **`delay()`**).

Another way to think about it is that use of delay slows down how quickly the loop function runs. If we are running multiple processes, we probably want the loop function to operate as quickly as possible.

Thus, we need a way to get around using delay when programming for multi-process programs.

The alternatives you use will depend on why you are using the delay ( ) function. Here are some examples:

1. Control DURATION BETWEEN TWO STATEMENTS.
(eg - piezo-speaker/tone example)

```
// -----------

void setup() { }

void loop()
{
  digitalWrite(pin1, HIGH);  // fix this!!

}
```

2. Control duration of a STATEMENT CURRENTLY BEING EXECUTED.

```
if(tnow - to <= duration)
{
  digitalWrite(pin1, HIGH);
```

```
}
else
{
  digitalWrite(pin1, LOW);
}
```

3. REPEAT A STATEMENT AT A GIVEN FREQUENCY (use an "if")

This was the case for the blinking LED (use "if" and toggle a byte?)

An IF statement is a good way to do this. The "if" statement would test whether a DURATION has exceeded some threshold. If so, then execute some code. Here you must access some of the Arduino timer functions (millis(), micros()). This will work fine as long as you are only doing one thing.

The challenge is to achieve these goals in the context of a repeating loop function.

One step better would be to place the "if" code into a FUNCTION. With a function, the interval and start time would be passed to the function as an argument. Within the function those variables need to be assigned to STATIC variables so their values are not "forgotten" when the function is executed the next time through the loop. The use of static variables in the function may limit the number of times you can call the function. If you wanted to blink 10 LED's using that same function (which you should be able to do - that's one of the reasons to use functions: re-usability), you'd need to declare 10 static variables. If you don't know how often you will call the function, then you don't know how many static variables to declare. In short, the use of static variables in the function can limit its reusability.

An even better approach would be to use CLASSES. Unfortunately this is more advanced programming. CLASSES are user-defined data types that have MEMBER VARIABLES and MEMBER FUNCTIONS. The member functions can manipulate member variables. A class DEFINES a new data type - it does not declare variables of that data type. After creating a class, you must declare an INSTANCE of the class. Analogous to this would be declaring a standard data type like an INT (e.g., int varName;). It's the same idea except with classes you are defining your own data type! (like MyInt).

TIMER & THRESHOLD

One way you can find online is to access Arduino's timer function and test whether you've passed a threshold.

USE SINE WAVES

Arduino has built-in trig functions (e.g. `sin()`) that you can use to pulse at a regular rate. Frequency is easy to program, but it's more difficult to control "duty cycle" (percentage of time high) with this method. But it is easy to control multiple pins with this method.

USING FUNCTIONS

A better way to "blink" something is to create a re-usable function. However, timer and threshold methods require the use of STATIC local variables. These are variables that are remembered after the function ends. This requires that you know how many times the function will be used (not good!). So this can work but has limitations. Another limited option is to use global variables (again, you'd need to know how many times you'd call the function, limiting its re-usability).

USE CLASSES

The best way is to create a new data type (a CLASS) that has member functions and variables. You then declare an INSTANCE of that class each time you want to blink something. Creating a class is tantamount to creating your own data type (e.g., an "int_ct"). This is what we do when we declare a variable like an int or a float. We can do this as many times as we want! The main problem here is that classes are more advanced programming.

```
while(timer < threshold) {}
```

CHAP 8 - LOOP SPEED

How fast is the loop function in Arduino? There is no built-in control for the frequency of the loop function. It runs as fast as possible. Thus, the frequency that it runs depends on the code that is placed inside of it. In theory, it could run as fast as 16 MHz, the clock speed of the Arduino CPU. In practice, every statement in the loop function will consume clock cycles, slowing the loop down.

Calls to the Serial Monitor slow down loop quite a bit.

We have tested loop to run at about 167 Hz (~ 6 ms/cycle) when calling "Serial.print".
Toggling digitalWrite( ) runs at about 120 Hz.

We have gotten speeds of up to 270 kHz to just increment a variable, with just periodic calls to the Serial monitor.

8.1 CHECKING LOOP SPEED

The follow code may help us get a sense of how fast the loop function can run. Online you will find users controlling registers directly, which results in a much faster code. Here, we will do something simpler. We will simply increment (++) an index each time through the loop and display how much got added every once in awhile. The Serial.print command is slow, so we will only run it once in awhile.

The result seems to show 270 kHz, not bad, but about 60x slower than 16 Mhz.

CODE

```
//loopseed.ino ---------------------
// test frequency of loop () fcn (don't use delay!!)
// getting 270kHz (is that right?)

long tlast = 0;
long loops = 0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  long tnow = millis();
  loops++;                        //tally loops

  if (tnow - tlast > 1000)
  {
    Serial.print("Loops per sec: ");
    Serial.println(loops);        // after 1 sec, display tally
```

```
    tlast = tnow;
    loops = 0;
  }
}

//end ------------------------------------
```

8.2 LOOP SPEED WHILE TOGGLING I/O PIN

Now let's check loop speed while toggling an I/O pin. Again - don't use delay(). Here we try to program better by placing the Serial printing in a function. Toggling the pin is also placed in a function. Here we seem to get 71.5 kHz (with the SM done every 1000 ms). If we increase the SM call to every 50 ms, then the speed drops to 3.3 kHz.

CODE

```
//loopseed_toggle.ino ---------------------
// test loop speed while toggling i/o pin... get 71.5k Hz at SM call of 1000 ms

long tlast = 0;
long loopcount = 0;
const int ledpin = 13;

void setup()  // -------------------------
{
  Serial.begin(9600);
  pinMode(ledpin, OUTPUT);        // set as output
}

void loop()  // -------------------------
{
  togglepin(ledpin, 200.0);
  printSM(50);

  loopcount++;
}

void togglepin(int pin, float freq ) // -------------
{
  static boolean pinstate = LOW;         // boolean = low or high
  static unsigned long tlast = 0;
  static float dur;
  unsigned long tnow = millis();        // get current time

  dur = (long)  1000.0/(2.0 * freq);    // convert freq to interval

  //toggle pin
  if (tnow - tlast > dur)
  {
     pinstate = !pinstate;
     tlast = tnow;                       //reset timer
  }
  digitalWrite(pin, pinstate);
}

void printSM(int dur) // -----------------
```

```
{
  long tnow = millis();
  if (tnow - tlast > dur)
  {
    Serial.print("Loops per sec: ");
    Serial.println(loopcount);

    tlast = tnow;
    loopcount = 0;
  }
}
//end -------------------------------------
```

CHAP 9 - MATH FUNCTIONS

The Arduino software has numerous built-in math functions

9.1 PI

The value of PI is built into the Arduino software. There is no need to do a #define.
It should by typed in all capital letters.

9.2 SINE & COSINE

Sine and cosine functions are built into the Arduino IDE.

The code below would implement a sine wave in Arduino.

```
float sig;
float freq = 1.5;       // in Hz
float t = millis()/1000.0;
float sig = sin(2 * PI * freq * t);
```

CHAP 10 - LIBRARIES

The Arduino software has numerous LIBRARIES that have specialized functions for doing a variety of tasks. Many of these libraries are installed when the Arduino IDE software is installed. Other libraries may be

installed afterwards (Menu > Sketch > Include Library > Manage Libraries ...). Still others, written by various users, may be downloaded from the internet.

Library files and folders generally reside in the following directory:

C:\Program Files (x86)\Arduino\libraries

10.1 SERVO LIBRARY

The SERVO library allows you to control servo motors. To access the library you must "include" Servo.h (`#include <Servo.h>`) allows our program to access the "Servo" library of functions.

"Servo" is a class – a data type with certain associated attributes and functions. "`Servo servoL`" creates an "instance" of the Servo class, much like we do when we write "int i" (which creates an instance of the class "int"). The instance has access to the various functions of that class (in the case of "Servo", there are functions like "attach", and "writeMicroseconds".

Below is a summary of some of the "Servo" class functions.

`servoL.attach(pin)` associates the servo signal to a specific pin

`servoL.writeMicroseconds(pulseWidth)` sends a pulse to the attached pin for a certain time.
If "1500" is the argument, then the pulse width is 1500 microseconds. A microsecond (us) is one-millionth of a second, so there are one million micro-seconds in one second. 1500 us also = 1.5 ms

`servoL.detach(pin)` stop servo motor operation

10.2 INCLUDE LIBRARIES

Arduino IDE provides an easy way to access a number of libraries. Many libraries are installed on your computer when you install the Arduino software. To access those libraries you must add the #include preprocessor directive to your code.

Sketch > Include Library > (select libarary)

This will add the "#include <library.h>" preprocessor directive to your code.

10.3 ARDUINO LIBRARY MANAGER

There are many libraries that are not installed with the Arduino software. However many popular libraries may be found using the Arduino LIBRARY MANAGER.

Tools > Manage Libraries ...

OR

Sketch > Include Library > Manage Libraries...

A long list of possible libraries appears. You can then select the desired library and it will be installed onto your computer.

10.4 ZIP LIBRARIES

Not every library will appear in the Arduino Library Manager. Perhaps you purchased a piece of hardware that uses a more obscure library. You may have to obtain the library on the hardware producer's website. Often you will download a zip file. Then you install the library using the following:

Sketch > Include Library > Add .ZIP Library

When doing it this way Arduino installs it in another directory:

C:\Users\(UserName)\Documents\Arduino\libraries

Oddly the Arduino software insists on the imported library being a zip file. If you have an extracted folder instead of a zip file, you can copy the folder into the directory above. You will have to restart the Arduino software to get it to appear in the Arduino menus.

CHAP 11 - SERIAL MONITOR INPUT

We've used the SERIAL MONITOR (SM) to output a variety of variable values. The SM can also be used to enter data into our program, but it is trickier to do this.

https://www.youtube.com/watch?v=q416XXJDE_M

11.1 ENTERING NUMBERS IN SM

If you type a number into the Arduino Serial monitor, a number is NOT sent to the Arduino. Instead a "char" data type is sent that is ASCII (American Standard Code for Information Interchange) encoded. The standard ASCII table associates characters (including typed "numbers") with a specific number. For instance lower case "a" is 97. So if you type an "a" in the Serial Monitor, the Arduino actually receives a 97. What about numbers? Unfortunately the ASCII code for typed numbers are not the numbers themselves. For instance, the number "2" is associated with 50. If you type a "2" in the Serial Monitor, the Arduino doesn't receive a 2, it receives a 50!

CODE

```
//SM_input.ino -----------------------------
// Pass data to Arduino by typing it into the serial monitor

byte b;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  while(Serial.available() == 0)      // wait here until something is entered in SM
  b = Serial.read();
  Serial.print("I got ");
  Serial.println(b);
}

//end --------------------------------------
```

CHAP 12 - INTERRUPTS

If you place the statement **`digitalRead()`** into the loop function, you are using a technique called POLLING. Polling is easier to program, but it wastes CPU computations checking on an event that may or may not happen at that moment. A more efficient approach is to use INTERRUPTS. The Arduino is set up to be able to handle interrupts.

To illustrate polling vs. interrupts, let's consider the task of counting lightning strikes. With polling, you'd get up from chair every, say, 5 minutes to look out the window for this discrete event. It wastes your time and energy. Plus you could miss the event (lightning strikes when you are not looking). Instead interrupts alert you to the event (e.g., the thunder tells you that lightning has struck). You then stop what you're doing and tally the count. You don't waste your time and you will not miss the event. Interrupts are a bit trickier to program though. Note – polling could work okay if the loop ( ) function is cycling fast enough so you don't miss the event.

Interrupts with Arduino work when an event occurs at one of the digital pins. The hardware interrupts the CPU and has it immediately execute a special function called the interrupt service routine (ISR). The ISR should be a short and fast bit of code so the processor does not miss another important computation.

There are 3 types of interrupts: change, rising, and falling. ISR's cannot return values or take parameters. The delay () and millis() don't work inside ISR's. The delayMicroseconds ( ) WILL work in ISR's. Any variables changed in the ISR must be declared with VOLATILE modifier. Only some pins have so-called "external interrupts" (on Arduino UNO pins 2 and 3).

For Arduino there are two types of interrupts: EXTERNAL INTERRUPTS and PIN CHANGE INTERRUPTS (PCI). External interrupts are a bit easier to program but there are only 2 pins on UNO (pins 2 and 3). Arduino Mega has 6 pins that work with external interrupts. PCI can work with all pins, but it is harder to program PCI's.

CHAP 13 - I2C COMMUNICATION

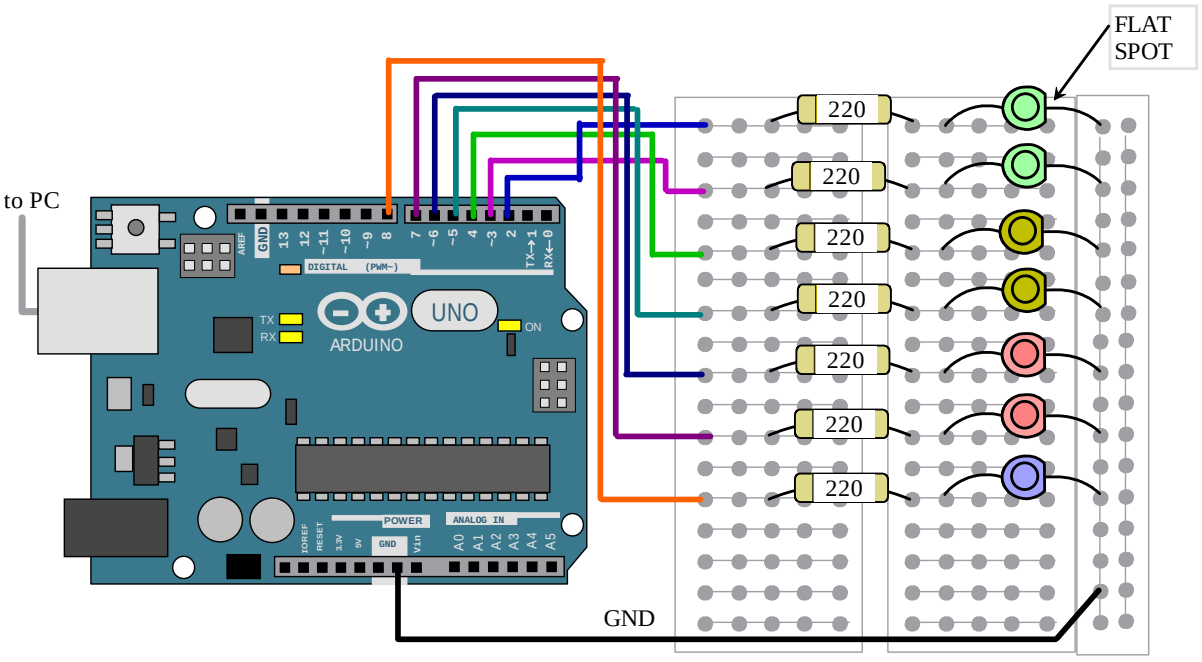
The i2C (inter-integrated circuit) communication protocol ("i-squared C") is a serial communication protocol. i2C allows a master controller to control or sense many different SLAVE devices using just 2 lines of communication. It can control up to 112 devices when using 7-bit addressing. It can control up to 1024 devices with 10-bit addressing. i2C can be used to have a "master" Arduino control many "slave" Arduinos. It can also be used to have an Arduino control another hardware device (the slave).

Each device has a pre-set ID (usually in hex form). There are 2 lines of communication. The serial clock line (SCL) synchronizes the data transfer. The serial data (SDA) line carries the data.

Examples of hardware we have controlled with Arduino using i2c include: displays and inertial motion units (MPU 6050). Displays are often controlled with i2C because there are more pixels on the display than there are I/O pins on Arduino. Without serial communication, each pin on Arduino would have to control an individual pixel on the display (this is not possible).

CHAP 14 - EXAMPLE ARDUINO EXERCISES ****

CHAP 15 - CHASING LEDs & RGB LED



Here we will learn to control the lighting of several LED's in a pattern and to control the lighting of an RGB LED, which has all 3 colors in one LED (red, green, blue).

PARTS LIST

1.) Computer, Arduino, USB cable
2.) Solderless breadboard
3.) (7) Resistors, 220 Ω

4.) (7) LED's
5.) (1) RGB LED

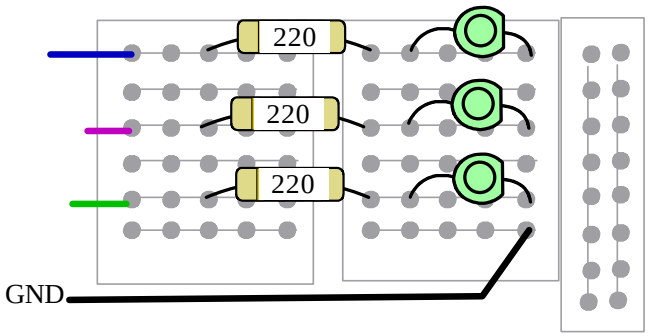
15.1 LED CHASE REBOUND

EXERCISE

Let's take a series of 7 LEDs and make a light show! Let's get the LEDs to "rebounding chase" where the light propagates to the right, then to the left, then to the right, over and over.

HARDWARE & CONNECTIONS

Wire up LEDs and current-limiting Rs onto a breadboard as in last lab, except now there's more LED's.
Power each LED to a separate I/O pins on Arduino.
All LED's must be hooked to the same GND on Arduino.



EXERCISE

```
// LEDchase_rebound.ino ----------------
// Create chasing LED (L > R > L > etc.)
// credit to Instructables.com

byte ledPin [] = {2, 3, 4, 5, 6, 7, 8};
int noLEDs = 7;
int duration = 60;
int currentLED;
int direction = 1;
unsigned long tlast;


void setup() // ------------------------------------------------
{
  for (int i = 0; i < noLEDs; i++)
  {
     pinMode(ledPin[i], OUTPUT);            //set all pins to output
  }
}

void loop() // -------------------------------------------------
```

```
{
   if ((millis() - tlast) > duration)
   {
      changeLED();
      tlast = millis();
   }
}

void changeLED() //---------------------------
{

   for (int i = 0; i < noLEDs; i++)
   {
      digitalWrite(ledPin[i], LOW);          //make all LEDs off
   }
   digitalWrite(ledPin[currentLED], HIGH);   //except current LED on
   currentLED += direction;                  //index to next LED

   if (currentLED == noLEDs)
   {
      direction = -1;                        //chg dir if at last LED
   }

   if (currentLED == 0)
   {
      direction = 1;                   //chg dir if at first LED
   }
}

//end ------------------------------------------------------------
```

OUTPUT / RESULT

A series of LEDs should light in a "chasing" pattern: left to right, then right to left; and so no.

15.2 LED ONE-WAY CHASE

EXERCISE

Alter the code above so that the LED's light in only one direction (L or R, your choice). If going right, once the right-most LED is lit, the lighting jumps to the left-most LED.

HARDWARE & CONNECTIONS (no change vs. above)

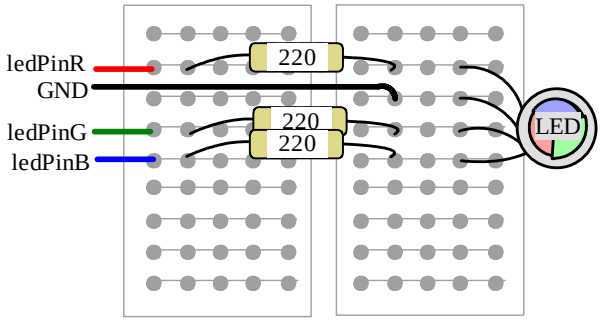
CODE

Your turn! See if you can figure it out!

OUTPUT / RESULT

As indicated above, LEDs "chase" in one direction, and then start over once they reach the end.

15.3 RGB LED



EXERCISE

Get the each color of the RGB (red-green-blue) LED to blink at different rates in a fading pattern.

HARDWARE & CONNECTIONS

Wire up LEDs and current-limiting Rs onto a breadboard.
Power each LED to a separate I/O pins on Arduino. GND must be hooked up too.
(note the diagram below is not complete - you need to figure out the Arduino connections on your own!)
Each LED pin must be a PWM pin (denoted with ~)

CODE

Here we use the sin() function because it's easy to adjust frequency using it. We also need to use the map() function which will map sinusoid values (-1 to 1) using linear interpolation to analog out values (0 to 255).

```
//lab7c_RGB_LED.ino ----------------
//Blink R,G, B on RGB LED at different freq's

int ledPinR = 9;                //must use pwm pins to fade
int ledPinG = 10;
int ledPinB = 11;

float freqR = 0.3;
float freqG = 0.5;
float freqB = 0.8;


void setup() // -------------------------------------------------
{
  pinMode(ledPinR, OUTPUT);           //set all pins to output
  pinMode(ledPinG, OUTPUT);           //set all pins to output
  pinMode(ledPinB, OUTPUT);           //set all pins to output
}
```

```
void loop() // ------------------------------------------------
{
  //use 3 sine waves!
  blinkLED(ledPinR, freqR);
  blinkLED(ledPinG, freqG);
  blinkLED(ledPinB, freqB);
}

void blinkLED(int LEDpin, float freq) // -------------------------
{
  float t = millis()/1000.0;
  float sig = sin(2*PI*freq*t);
  int sig2 = (int) 100.0 * sig;              //range -100, 100
  int ledLevel;
  ledLevel = map(sig2, -100, 100, 0, 50);

  //Serial.println(ledLevel);

  analogWrite(LEDpin, ledLevel);
}


//end ---------------------------------------------------------------------
```

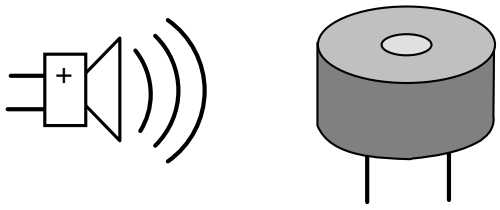
OUTPUT

The different colors (red, green, blue) of an RGB LED should fade in and out at different frequencies.

CHAP 16 - HELPFUL CODE

Here we show some other code that you may use with Arduino

16.1 PIEZO-SPEAKER



Piezo-electric materials convert electricity to mechanical motion and vice-versa. A piezo-speaker will take frequency signals from Arduino and produce tones. Many piezo-speakers are designed to play 4.5 kHz tones for smoke alarms, but it can also play a variety of audible tones usually best in the 1 kHz to 3.5 kHz range. The function tone (pin#, freq, duration) is often used with the piezo-speaker. This command is "asynchronous" meaning that once it executes it can work in the background even though other commands may be subsequently executed. Delay ( ) does not stop it either.

Tone cannot be run on more than 1 pin at a time on Arduino. Also you cannot analog write to pins 3 and 11 while using tone ( ) (they use the same timer).

Example code for a piezo-speaker is below.

```
//Play1note.ino --------------------------
//pick one note from an array, display value in SM, play on piezo speaker

int note[] = {1047, 1147, 1319, 1397, 1568, 1760, 1976, 2093};
void setup()
{
  Serial.begin(9600);
  Serial.print("note = ");
  Serial.println(note[3]);
  tone(4, note[3], 500);
  delay(750);
}
```

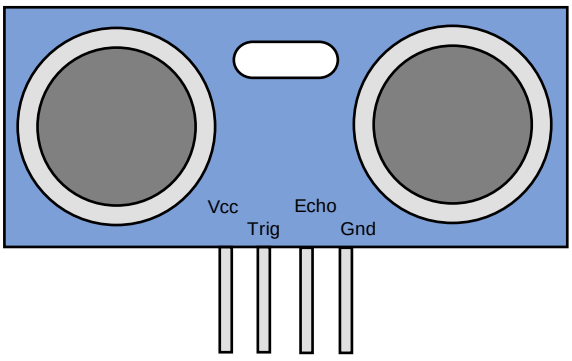
There are two options for the tone function as shown below. The first option may be used if you want the tone to go on indefinitely. The second option allows you to specify the duration of the tone (in ms). The delay ( ) command after tone is NOT necessary for the tone command to finish - it will finish with or without delay(). The delay() function is placed there to delay the NEXT command from executing before tone( ) is completed. This may or may not be what you want.

```
tone (pin, frequency)                  // tone keeps going
//OR
tone (pin, frequency, duration)

//eg
tone (4, 3000, 1000) // play tone on pin 4 at 3 kHz for 1 sec

delay (1000);                          //delay NEXT cmd so tone() can finish
```

16.2 ULTRA-SONIC SENSOR



ULTRA-SONIC SENSORS sense distance by sending out a ultrasonic pulse of sound and measuring the time it takes for that sound pulse to be received back. Distance can be inferred from the time it takes for the pulse to be received. This sensor is commonly included in many Arduino kits. While the two cylindrical parts look like eyes or speakers, they are not. One is an ultrasound transmitter, and the other a receiver. There are 2 sensors on the market. One is by Parallax called the "PING" sensor ($30). The other is the HC-SR04 ($4). The PING sensor is more expensive but it works with smaller targets and only uses 1 I/O pin. The HC-SR04 has more

erroneous signals with smaller targets and uses 2 I/O pins. I will refer to theses sensors as "PING SENSORS" from now one, even though you may be using the HC-SR04.

The PING sensor sends out an ultrasonic pulse of sound at 40 kHz (we can't hear that).

SPECS

Vcc	5 VDC	Gnd	0 VDC
Frequency	40 kHz		
Range	2 cm to 4 m	Accuracy	+/- 3 mm

The ping sensor will generate an 8-cycle ultra-sonic (at 40 kHz) burst (or pulse) in response to a 10 us pulse on the trigger pin. The receiver will wait to receive the reflected sonic burst and the sensor will output a pulse on the echo pin that is equal to the time between transmit and receive (in micro-seconds, us).

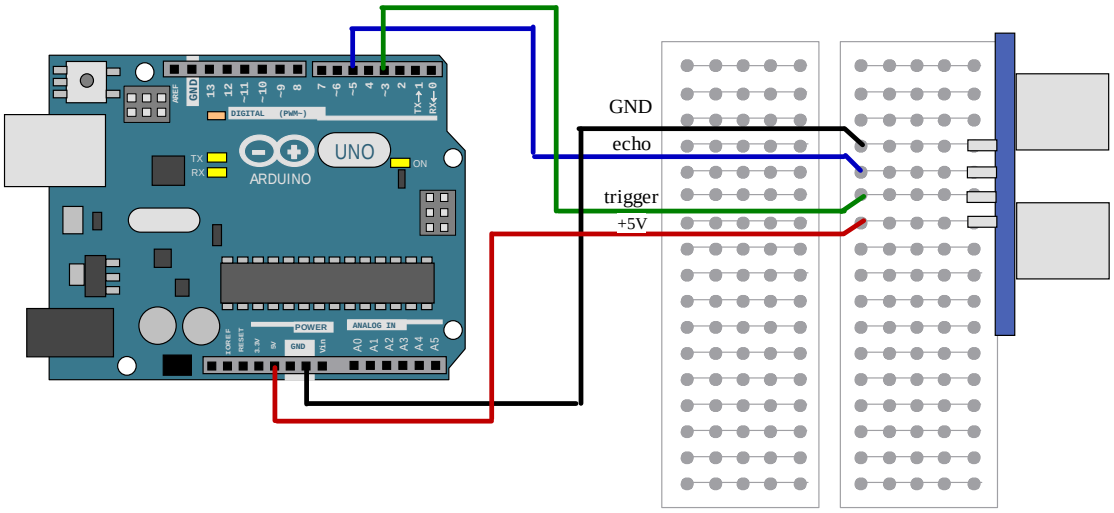
Watch this informative video:
https://www.youtube.com/watch?v=ZejQOX69K5M

In the code below is the statement

```
distance = duration * 0.034/2;
```

The "0.034/2" is there because sound travels at .034 cm/us. You divide by 2 because sound is traveling twice the distance you want to display (it goes out, reflects off a surface, and returns to the sensor). The result is to display distance in cm.

WIRING



PING DISPLAY DISTANCE

In this exercise the ping sensor will sense distance and Arduino will display the result (in cm) on the serial monitor.

```
// Ping_Distance_display.ino ---------------
// display distance (in cm) on SM using ping sensor
```

```
const int trigPin = 9;
const int echoPin = 10;

long duration;
int distance;


void setup()
{
  pinMode (trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);

  duration = pulseIn(echoPin, HIGH);
  distance = duration * 0.034/2;

  Serial.print("distance: ");
  Serial.println(distance);
}
```

DISANCE MAPPING

You can mount the ping sensor onto a servomotor and have the servo sweep over a range of angles as the ping sensor collects distance data. This allows you to map distances over a range of angles. In this way you can get a mobile robot to navigate through a maze or around a course of obstacles.

TODO